

Java Basics

Introduction to Module 1:

In our first module, you'll learn the basics of the Java programming language. We'll focus on building your comfort and familiarity with the syntax of the language while giving you the building blocks that you will use throughout the rest of the course.

IN THIS MODULE WE'LL COVER...

- Writing your first Java program
- The different parts of a Java program
- Printing text to the console
- Organizing your code using methods
- How Java views and stores basic types of data
- Different methods of manipulating data
- How to create programs that allow a user to interact with the software

What is Java?

<<Video>>

Learning to code is quickly becoming a requirement for any field of work. There is almost an infinite number of programming languages you can choose when learning to code and chances are you might have already tried some. In this course, we are going to be learning a programming language called "Java".

Java was written by Sun Microsystems in 1995 and has remained a popular and well-respected language for over 20 years. That is no small feat for a programming language. Despite being one of the older programming languages in use, Java remains one of the most popular for a variety of reasons:

- It's relatively easy to pick up
- It has great features for writing large and complex software solutions
- It has a ton of *libraries* with code ready for you to use

- There are plenty of resources for Java developers including books, courses, programs, and communities!

In this course, we choose to teach you Java because there are some very important aspects of writing code, and of the field of computer science, that Java does a really good job of handling. This gives us the opportunity to really dive into the most interesting conceptual ideas in building software solutions, and along the way, you will learn an industry standard language and method of writing code that is in high demand in the tech industry. These are some of the reasons many universities also use Java as their first programming language they ask students to learn.

Hello World

Hopefully you've already [downloaded and installed all the tools you need to develop in Java](#)

Once you have your environment all set up let's [write our first Java program- the traditional Hello World.](#)

HelloWorld.java

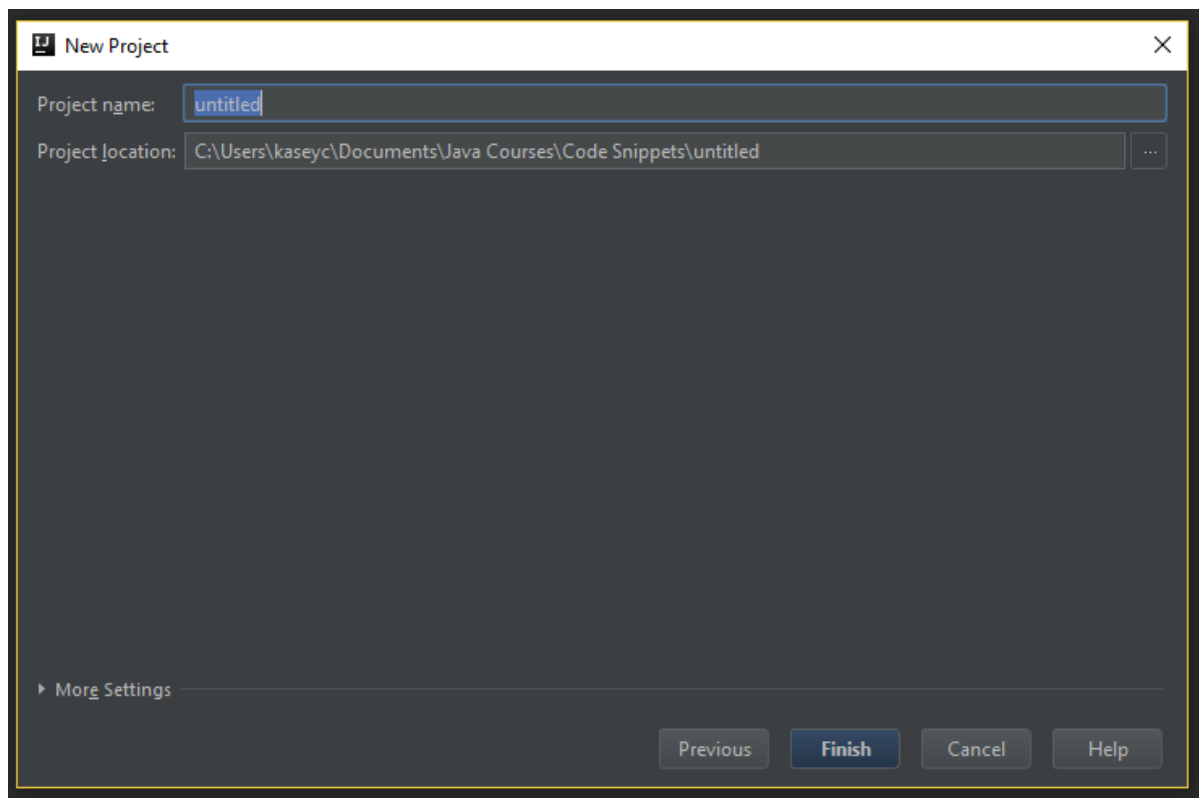
```
public class HelloWorld {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello World");  
  
    }  
  
}
```

<<Video>>

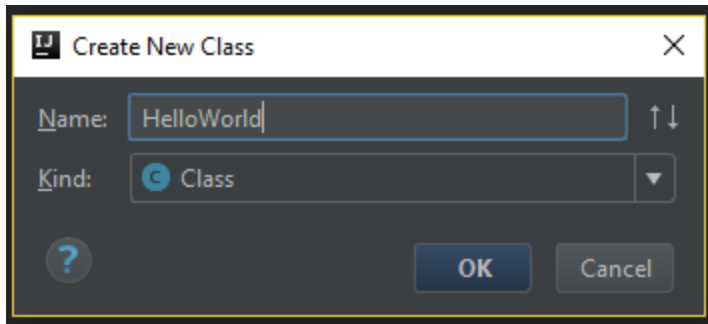
Step 1 - Create a new .java file in your "IDE" (Integrated Development Environment)

If you are using the program IntelliJ like we are, you will need to create a new project, then within that project create a new file under src that is of type "Java". When you make a project there are a couple different names you'll need:

- Project Name - this is what you enter when you are creating a new project with the IntelliJ dialog boxes. This will be the name of the folder where all your different project files will be kept. For example, you might call your project "Java practice" and put all your different code files in one project.



- Java File Name - this is the name of the actual file that will hold your code. It will be stored in the source or "src" folder within your IntelliJ project. This should follow the below java naming conventions.



- Class Name - this is the name of the class that you declare in your code. Java requires that this name **exactly match** the name of the .java file. In fact, IntelliJ will automatically name your class for you based on what you entered for the file name and generate code that looks like the following:

A screenshot of the IntelliJ code editor showing the contents of 'HelloWorld.java'. The code is as follows:

```
1  /**
2   * Created by kaseyc on 7/10/2017.
3   */
4  public class HelloWorld {
5  }
6
7  |
```

Java Naming Conventions

As you create new Java files and Classes you must give each a unique name. Java has its own "conventions" or suggested formats for how to create these names:

- No spaces
- Must start with a letter
- Capitalize the first letter of each new word

Step 2 - Write your code

Once you have a file that can understand Java code, it's time to add your code! Try adding the code at the top of this page to your Java class. You'll know if you've set it up correctly because certain words should change colors within your IDE. These are called "key words" and your IDE should highlight them to make it easier for you to read your code.

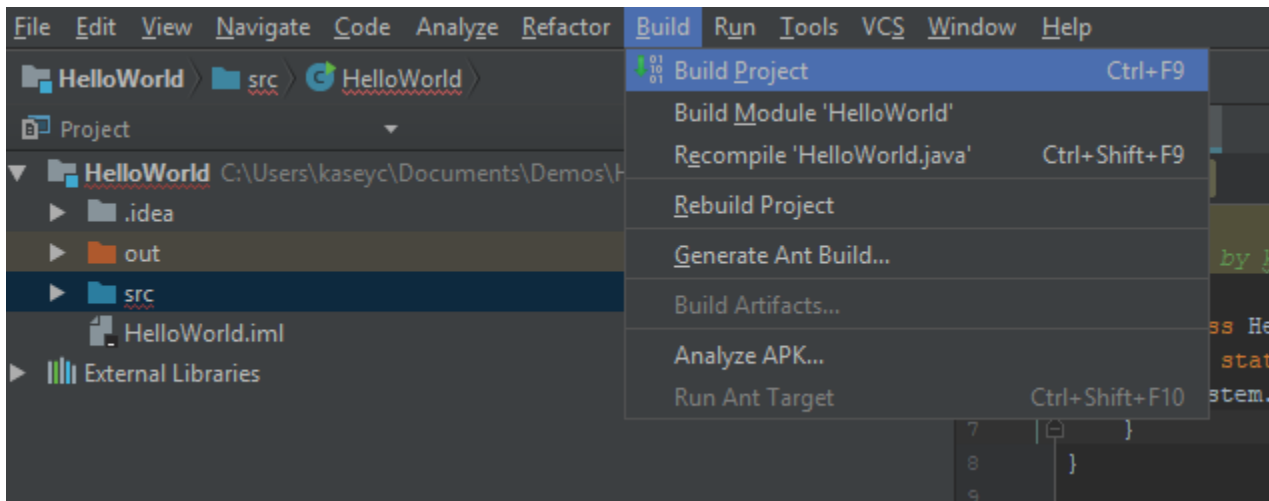
```

1  /**
2   * Created by kaseyc on 7/10/2017.
3   */
4  public class HelloWorld {
5      public static void main(String[] args) {
6          System.out.println("Hello World");
7      }
8  }
9
10

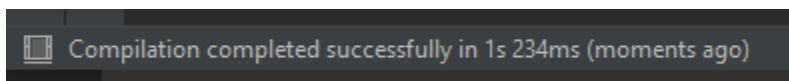
```

Step 3 - Compile/Build

If this is the first time you have compiled this file, you need to tell your program to "build" the project.



If you pay close attention to the bottom of your program screen you will see a message that tells you how long it took to compile your code, like this:



If you have "compiler errors" you will see red squiggly lines under the parts of the code the compiler doesn't like.

```

public static void main(String[] args) {
    System.out.println("Hello world")
}

```

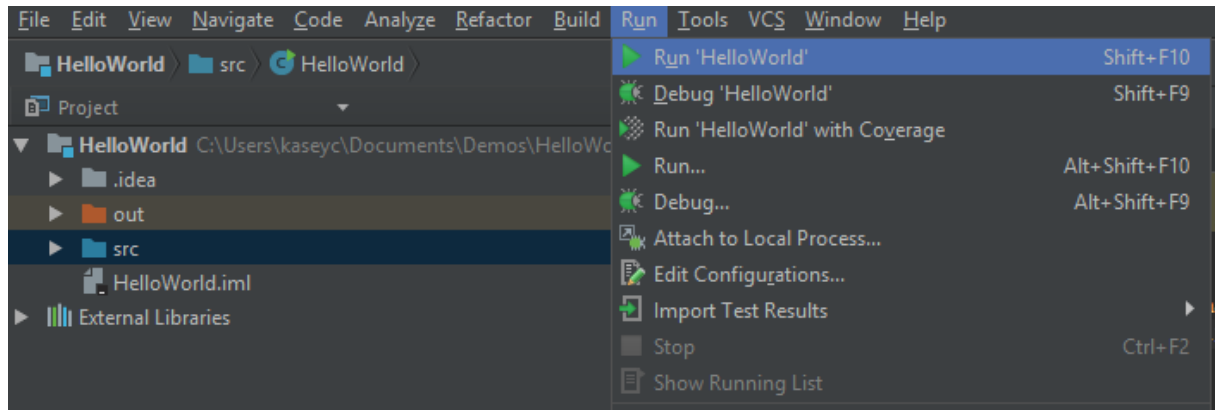
If you hover over those lines with your mouse it should tell you the message the compiler produces as to why it does not understand your code.

```
public static void main(String[] args) {  
    System.out.println("Hello world")  
}
```

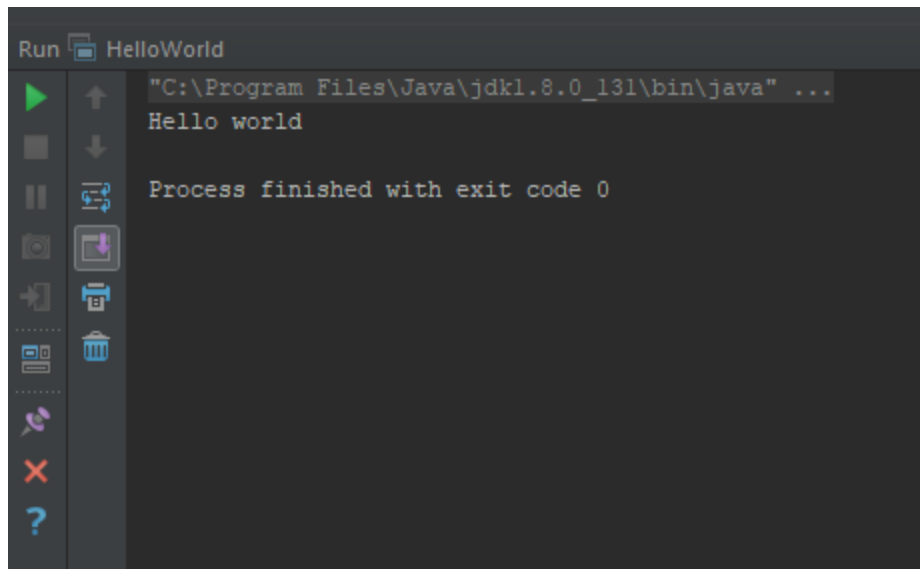
';' expected

Step 4 - Run

Once you have gotten rid of all your compiler errors, it's time to tell your computer to execute your code. This is called "running" your code. You can find the options to do this under the run menu in IntelliJ.



If your program runs successfully you should see the results in the console. In IntelliJ this will cause a window at the bottom to pop up like this:



The screenshot shows a dark-themed IDE window titled "Run HelloWorld". On the left is a vertical toolbar with icons for running, stepping through, and other debugging actions. The main console area displays the following text:

```
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...  
Hello world  
  
Process finished with exit code 0
```

Parts of a Java program

Anatomy of a Java Program

<<Video>>

There are generally three different pieces of a Java program

class: a program

method: a named group of statements

```
public class name {  
    public static void main (String [] args) {  
        statement;  
        statement;  
        statement;  
        ...  
    }  
}
```

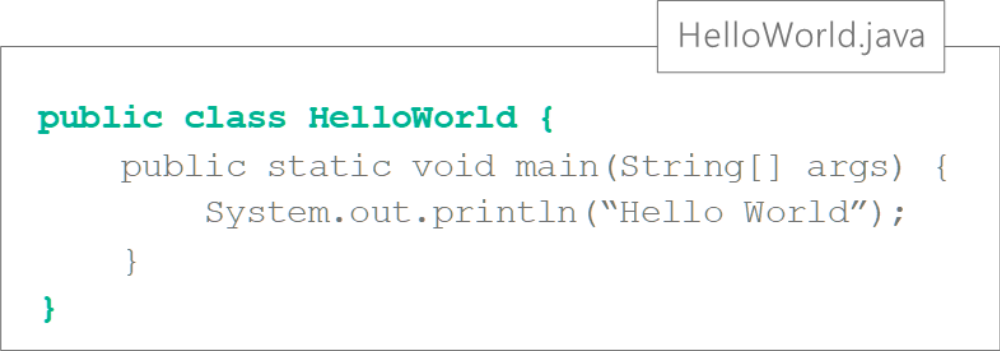
statement: a command to be executed

Class

The **class** is the outermost layer of the program. It starts with the keywords "public class" and then whatever the name of your file is.

```
public class YourClass {
```

The name of your class has to exactly match the name of your file, including capitalization. The class is enclosed by a set of curly braces or "{}" characters. All code you want to run should be placed between these two characters.



```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Main Method

The **main method** is the wrapper that tells your computer which lines of code to run. In Java it has a very specific series of keywords:

```
public static void main(String[] args) {
```

After the open curly brace, you should put each line of code you want the computer to run. Don't forget to add the closing curly brace after all your code. As you can see in the image it is convention to add a layer of indentation between the main method and the class, this shows that the main method is inside the class.

HelloWorld.java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Statements

The **statements** are the lines of code for the computer to run. Each line is called a "statement" and should end with a semi-colon(;). You can have as many statements as you want, the computer will run each line from top to bottom one after the other. Again, as these are inside the main method all statements should be one level of indentation in.

HelloWorld.java

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

Comments



```
1  /*
2     This program demonstrates the usage of different comment formats.
3     This top comment often describes the entire program, and this typically multiple lines
4     It is best practice to use a multiline style comment markets for a comment such as this.
5  */
6  public class LearningComments {
7      public static void main(String[] args) {
8          System.out.println("executable statements"); //this is a single line comment
9          System.out.println("executable statements"); //these are great for short, direct comments
10     }
11 }
```

Comments are notes written in the source code by the programmer to describe or clarify the code. They are ignored by the Java compiler so they are intended only as messages to the humans that will read the code files.

<<Video>>

LearningComments.java

```
/*

multi line comments are placed between the slash and asterisk markers

*/

public class LearningComments {

    public static void main(String[] args) {

        // single line comments are placed after double forward slashes

        System.out.println("Hello World");

    }

}
```

It is recommended that you add comments...

- at the top of any Java class to describe its overall function
- at the top of a "method" or a small subsection of code to help explain its contribution to the overall solution

- to explain what is happening when you have particularly complicated bits of code
- to isolate lines of code when looking for issues without having to delete code

There are two different ways to add comments

Single line comments

// comment text, on one line

Multi line comments

/* comment text

You can have as many lines as you like

Between these two indicators */

Basic Java Commands

Strings And Printlns

Strings

<<Video>>

In Java, a **String** is **any sequence of characters** – letters, digits, spaces, punctuation, and other “special characters”. Here are some examples of Strings defined in Java:

String	Explanation
"Hello, world"	You used this String in the previous section! It has 12 characters. The space between the comma and “w” is also counted as a character. The double quotes are not part of the String, they define where the String starts and where it ends.
"12234"	This is a String, not an integer number because it’s enclosed in double quotes.
"!"	This String is just a single character: an exclamation point.
">< _ ^\$#@^"	This is a String with 10 special characters in it.
""	This is a special String: an “empty String” that contains NO characters at all. The two double quotes are right next to each other, without even a space between them.

Escape Sequences

What if you want to create a String that contains a double quote character? Tricky! If you wrote "", Java would be confused. The computer can't tell if "" is an empty String followed by a double quote, or a String consisting of a double quote character. Java includes a way for you **to clarify this ambiguity with**

what is called an “**escape sequence**”. When Java sees a **backslash** in a String this tells Java to “**escape**” or skip the next character's function and simply add it to the String. Here are the escape sequences we’ll use in this course:

escape sequence	Name	Example	String as printed
"	Double quote	"John \"JJ\" Doe"	John "JJ" Doe
\	Backslash	"Use \\, not /"	Use \, not /
\t	Tab	"1\titem"	1 item
\n	New line	"line 1\nline 2"	line 1 line 2

Printlns

Computers wouldn’t be very useful if they couldn’t communicate with people and one of the simplest ways they do that is to “print out” text for us to read. To produce String based output we use the **println statement**. We’ve already used this when we wrote our “Hello, world!” program. This is a type of computer “output”. The place you look on your screen for text output from a Java program is called the “**console**” – a historical reference to the big desk-like things that the first generation of computers used.

Java prints text out to the console using a **System.out.println statement**. Here’s the one you used in your HelloWorld program:

```
System.out.println("Hello, world!");
```

It’s kind of wordy, but the words do make sense in a way:

- **System** essentially refers to the computer as a whole

- **out** refers to the standard output device: the console
- **println** indicates that you want to “print” as a complete line (ln) of text

Our statement above will print *Hello, world!* to the console area on your screen. It also advances to the next line on the display, just as if you typed it and then hit the Enter (or Return) key.

You can also use the `println` statement to produce an empty line.

```
System.out.println();
```

Prints

Sometimes you may want to use several print statements and have all the output produced on the same line. For this, you can write `print` instead of `println`. For example, these statements:

```
System.out.print("Hello, ");
```

```
System.out.print("Joe");
```

```
System.out.print("! Hello, ");
```

```
System.out.print("Jane");
```

```
System.out.println("!");
```

...would print out:

Hello, Joe! Hello, Jane!

... and advance to a new line only after the last exclamation point. That is because you can see that the first 4 calls leave off the "ln" at the end of `print`. These are simply "prints" instead of "print lines" and do not add the carriage return or "Enter" at the end of a line of output.

Finally, you can also print multiple lines with a single `println` statement by using the new line escape sequence from above:

```
System.out.println("line 1\nline 2\nline3");
```

... prints out these three lines:

line 1

line 2

line 3

Methods

<<Video>>

Divide and Conquer

Most useful computer programs are much longer than our five line "Hello World" program. Many programs are HUGE – literally tens of millions of lines of code. No human can possibly understand all 10 million lines of a program all at once – in fact, most programmers will admit that they can only focus on at most a couple hundred lines of code at a time, and experienced programmers prefer chunks of no more than 50 lines or so (about what you can see on a screen at a one time). To be able to create large programs, we have to be able to break the job down into small, manageable parts.

The process of breaking a task down into smaller parts is called "**procedural decomposition**", and almost programming languages provide a way to do this. Java does it using what it calls "**methods**". Our Hello World program had a single method called main, but most Java programs have several methods. In general, you should break your program up into methods when:

- You can break a task into distinct non-trivial subtasks in order to understand it better. For example:

Fly airplane 

1. Take off
2. Fly
3. Land

You could even break one or more subtasks down further:

Fly airplane

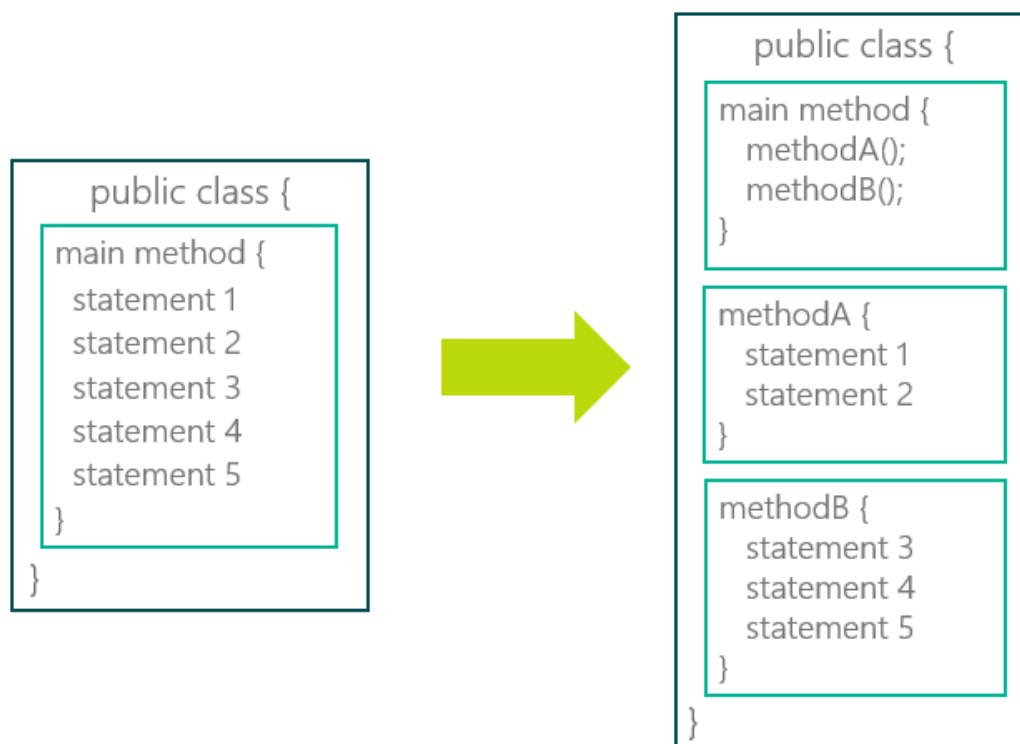
1. Take-off
 1. Push back from gate
 2. Taxi to runway
 3. Increase speed until off ground
 4. Climb to cruise altitude
2. Fly

3. Land

- You have "**redundant code**": unnecessarily repeated code that appears more than once in your program. It's OK to use methods to eliminate redundancy for even a single, complicated line. That way if you ever have to change it, you only have to change it in one place!
- Your method is getting bigger than 20-30 lines of code; look for groups of statements that are related pull them out into a method.

When you decompose a program into methods, examine each method to make sure it is doing a single, specific thing – not a random collection of unrelated things. Try to capture what the method is doing in a few words like we did in the airplane example above. If you can't describe your method's task in a few words, maybe it needs to be broken down further.

Ideally, once you have organized your program with methods, your main method turns into a list of method calls, each that handle their own small piece of the task to be performed. Thus your main method turns into a sort of outline of what your code is doing, making it easier for someone reading your code to quickly understand what it does.



Method names

Every method in a Java program has its own name. In Hello World, the single method's name was main:

```
public static void main (String[] args) {
```

main is a special method name in Java – it's the method Java will use to execute your program. Since this is a special method, you cannot use the name "main" for any of your other methods.

As the purpose of a method is typically to accomplish a task, it's a good idea to start a method's name with a verb: calculate, report, print etc... Followed by whatever item that task is performed on: calculateAge, reportResults, printSummary etc...

The Java convention is for method names to start with a lower case letter, and use uppercase letters to start each following word with no spaces. This is called "**camel case**": the uppercase letters are the camel's humps! Here's how the methods for our flying example should look:

flyAirplane

takeOff

pushBackFromGate

taxiToRunway

increaseSpeedUntilOffGround

climbToCruiseAltitude

Writing a Method

The first line of a method is called the "**method header**" and it contains some specific keywords:

```
public static void myMethod() {
```

Each of these keywords have specific functions, and we'll learn about some of the other keywords later in this course. For now, we're going to keep it simple so all our methods will be public, static, and void. Here is the syntax for creating your own method including the method header:

```
public static void methodName () {
```

statement 1

statement 2

...

statement n

}

This is called "**defining**" a method, and you should place your methods after the main method. This means they should be placed after main's closing curly brace, but before your class's closing curly brace, like so:

```
public MyClass {  
    public static void main(String[] args) {  
        statements...  
    }  
  
    public static void myMethod() {  
        statements...  
    }  
}
```

You would use **your method** name in place of methodName, and enclose all the statements of your method inside curly braces {}.

Calling a Method

Just defining a method won't actually tell the computer to execute that code. To actually run the code inside your method, another method has to "**call**", or invoke, it by name. Each method invocation statement starts with the name of the method you want to execute, followed by a pair of parentheses finishing with a semicolon. Again using our flying example, here's how the method calls would look:

```
public static void main(String[] args) {
```

```
    takeOff();
```

```
fly();

land();

}
```

```
public static void takeOff () {

    pushBackFromGate();

    taxiToRunway();

    increaseSpeedUntilOffGround();

    climbToCruiseAltitude();

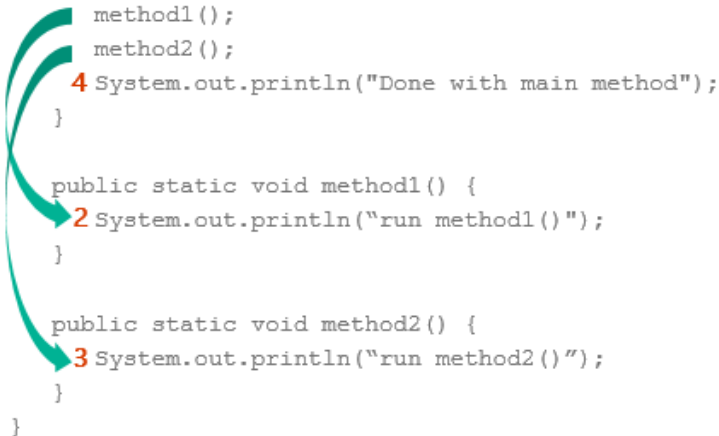
}
```

When you call a method, the computer jumps to that method definition, runs all the code inside that method, then returns to the line where it was called. We will discuss the details of how the computer runs methods in the next section.

```
public class LearningMethods {
    public static void main(String[] args) {
        1 System.out.println("Start main method");
        method1();
        method2();
        4 System.out.println("Done with main method");
    }

    public static void method1() {
        2 System.out.println("run method1()");
    }

    public static void method2() {
        3 System.out.println("run method2()");
    }
}
```



output
1 Start main method
2 run method1()
3 run method2()
4 Done with main method

Control Flow

<<Video>>

It may seem like the work computers do is very complicated, but really they are just performing simple tasks very quickly. As you learn to program you are learning how to give computers instructions on how to carry out these tasks. On your journey to becoming a programmer it is important for you to understand how the computer will read and run your instructions. Here are the rules computers use to run your code we've encountered so far:

1. First, find the main method. **If your program doesn't have a main method to get it started, it won't run!**
2. Execute each statement in the main method from top to bottom, starting with the one right after the opening curly brace.
3. If the statement is a method call, jump to that method definition and execute each of the statements in that method from top to bottom.
4. When you reach the end of a method (represented by its closing curly brace), "return" to where you encountered the method call and continue with the next statement following the method call.
5. When you reach the end of the main method, end execution and close the program.

These rules control the "flow" of our program from one statement to the next, this is called the "control flow" of the program.

Let's see how this works by reading through the Java code below. Reading code is an important skill to learn – most software developers spend more time reading code that someone else wrote (or that they wrote a while ago!) than they do writing brand new code.

```
public static void main(String[] args) {  
  
    System.out.println("main method starting...");  
  
    message1();  
  
    message2();  
  
    System.out.println("...done with main");  
}
```

```
}
```

```
public static void message1() {  
    System.out.println("All of message1.");  
}
```

```
public static void message2() {  
    System.out.println("Start of message2.");  
    message1();  
    System.out.println("End of message2.");  
}
```

Step	<div>Code</div>
1	Find the main method
2	Do each statement in order
3	If it's a method call, find that method
4	Do each statement in order

Step	Code
5	When you reach the end of the method, return to the previous method and continue with the next statement there
6	If it's a method call, find that method
7	Do each statement in order
8	If it's a method call, find that method
9	Do each statement in order
10	When you reach the end of the method, return to the previous method and continue with the next statement there
11	When you reach the end of the method, return to the previous method and continue with the next statement there
12	When you reach the end of the main method, you're done!

You can think of method call statements as expanding your program copying the statements of each method in place of its method call. It's like the program we would have written if we

hadn't decomposed it into methods! Of course, for this very simple example, we could have just written it like this in the first place.

```
System.out.println("main method starting...");
```

```
System.out.println("All of message1.");
```

```
System.out.println("Start of message2.");
```

```
System.out.println("All of message1.");
```

```
System.out.println("End of message2.");
```

```
System.out.println("...done with main");
```

Here is some code that involves methods calling other methods:

```
public class ControlFlow {  
  
    public static void main(String[] args) {  
  
        System.out.println("start main");  
  
        method1();  
  
        method2();  
  
        method3();  
  
        System.out.println("end main");  
  
    }  
  
}
```

```
public static void method1() {  
  
    System.out.println("enter method1");  
  
    method2();  
  
    System.out.println("end method1");  
  
}
```

```
public static void method2() {  
  
    System.out.println("enter/end method2");  
  
}
```

```
public static void method3() {  
  
    System.out.println("enter method3");  
  
    method1();  
  
    System.out.println("end method3");  
  
}
```

```
}
```

Here is an image of how the computer would run each of the lines of code in proper order, starting from top to bottom. As you can see each time it encounters a method call it jumps over to that method, executes all the code until it reaches another method call then jumps again. Eventually, as it reaches ending curly braces the computer returns to the origin of the method call and picks back up where it left off.


```

public class LearningMethods {
    public static void main(String[] args) {
        1 System.out.println("start main");
        method1();
        method2();
        method3();
        11 System.out.println("end main");
    }
    ...
}

```

```

public static void method1() {
    2 System.out.println("enter method1");
    method2();
    4 System.out.println("end method1");
}

```

```

public static void method2() {
    3 System.out.println("enter/end method2");
}

```

```

public static void method2() {
    5 System.out.println("enter/end method2");
}

```

```

public static void method3() {
    6 System.out.println("enter method3");
    method1();
    10 System.out.println("end method3");
}

```

```

public static void method1() {
    7 System.out.println("enter method1");
    method2();
    9 System.out.println("end method1");
}

```

```

public static void method2() {
    8 System.out.println("enter/end method2");
}

```

output

```

1 start main
2 enter method1
3 enter /end method2
4 end method1
5 enter /end method2
6 enter method3
7 enter method1
8 enter /end method2
9 end method1
10 end method3
11 end main

```

1

2

3

4

5

6

7

8

9

10

11

12

Data Types and Variables

Data Types

<<Video>>

So far, the only data our programs have used are Strings of characters contained within double quotes, like "Hello, world!". Computers process many different types of data: numbers, pictures, sound, etc. Java is very strict about defining exactly what type of data it is using at all times.

Java defines a few very simple and efficient **primitive data types**. There are eight of these in total, but we're going to focus on just four of them:

Integer

Integers represent whole numbers. You may be familiar with integers from algebra class. Integers are numbers that are not fractions nor have decimals, for example, 422, -13 and 0. Java refers to these as type **int**. As computer memory is not infinitely big there is a limit to how big an integer you can store. The largest int that Java can represent is actually 2,147,483,647, and the smallest is -2,147,483,648. This is big enough for most purposes, but note that it's only about ¼ the population of Earth, so there will be situations in which using an int won't work.

Double

Doubles are real numbers like 3.14159, -2.718, and 2145.2731. You can also write these in a form of scientific notation; for example, 1.23e2 is the same as 123, and 921e-1 is 92.1. For historical reasons, Java calls these type **double**. These can represent much bigger and smaller numbers, but they are in general only approximations, not exact values as they can only hold so many decimal places. This can lead to rounding errors when you might not expect them!

Character

Characters are single characters such as letters, symbols and spaces. These are not just limited to english, but all languages. For example, A, X, 0, z, ?, and =, but also Ω, ä, Я, and many more! Each of these is a **char** in Java. When you define a char, enclose it in single quotes, like 'a'. String objects are made up of **char** characters. **NOTE:** 'a' is NOT the same as "a": the first is a char, the second is a String that contains a single char.

Boolean

Booleans are a type of data that can only have either two values: True or False. Booleans are the building block for logic based decisions, you can think of them as representing yes or no, on or off,



positive or negative, etc... Java calls these boolean values, in honor of George Boole who created a theory of algebra for numbers with only two values.

Here's a summary of these Java primitive data types:

Java keyword	Data Type	Examples	Min	Max
int	integer	422, -13, 0	-2^{31}	$2^{31}-1$
double	real number	-23.1, 14.56, 9.4e3	-1.8×10^{-38}	-1.8×10^{38}
char	one character	'A', '1', 'z', '%'	NA	NA
boolean	true or false	true, false	NA	NA

Variables

<<Video>>

What are Variables?

All of the data we've seen so far have been **literals** of various types: specific fixed values, like 13 or 1.3 or "abc". Most useful programs also need to use data whose values change, for example, like a count of points scored in a game, or the amount of time remaining. In Java there is a construct to hold data whose value changes, these are called **variables**. You can think of a variable as a named box (or cell) in memory that holds a value that the program can use or change and refer to by name.

myIntVar



myCharVar



This is similar to using variables instead of numbers in algebra – an algebra equation uses a variable to define what to do with any value the variable can contain. For example, the equation of the line $y = x + 2$ says that whatever the value of x is, y is 2 more than x . Thus x is a stand-in for a range of possible numbers.

Declaring and Using Variables

Before we can use a variable in Java, we must **"declare"** it. To do this you must tell Java what type of data you want your variable to store and give the variable a unique name within the given scope.

`dataType variableName;`

For example, to declare a variable of type String with the name "myString" we write:

`String myString;`

myString



Declaring the variable creates the space in memory for a value, to actually store a variable in that space you use an **"assignment statement"**. An assignment statement associates the given value to the named variable. The assignment statement starts with the name of the variable whose value we are setting, then an equals sign, then the value we want to set, and finally, a semicolon to mark the end of the statement:

`variableName = value;`



If we wanted to store the String "Hello, world!" in our variable from before

```
myString = "Hello, world!";
```

myString



"Hello, world!"

We can now use the variable anywhere we could have used the literal string. This statement would now print "Hello, world!" to the console:

```
System.out.println(myString);
```

If we set the variable to a new value and then repeat the same println statement, it will print out the new value "Hello yourself!" instead:

```
myString = "Hello yourself!";
```

```
System.out.println(myString);
```

Java allows you to assign an initial value to a variable when you declare it; this code combines the first two statements above:

```
String myString = "Hello, world!";
```

```
System.out.println(myString);
```

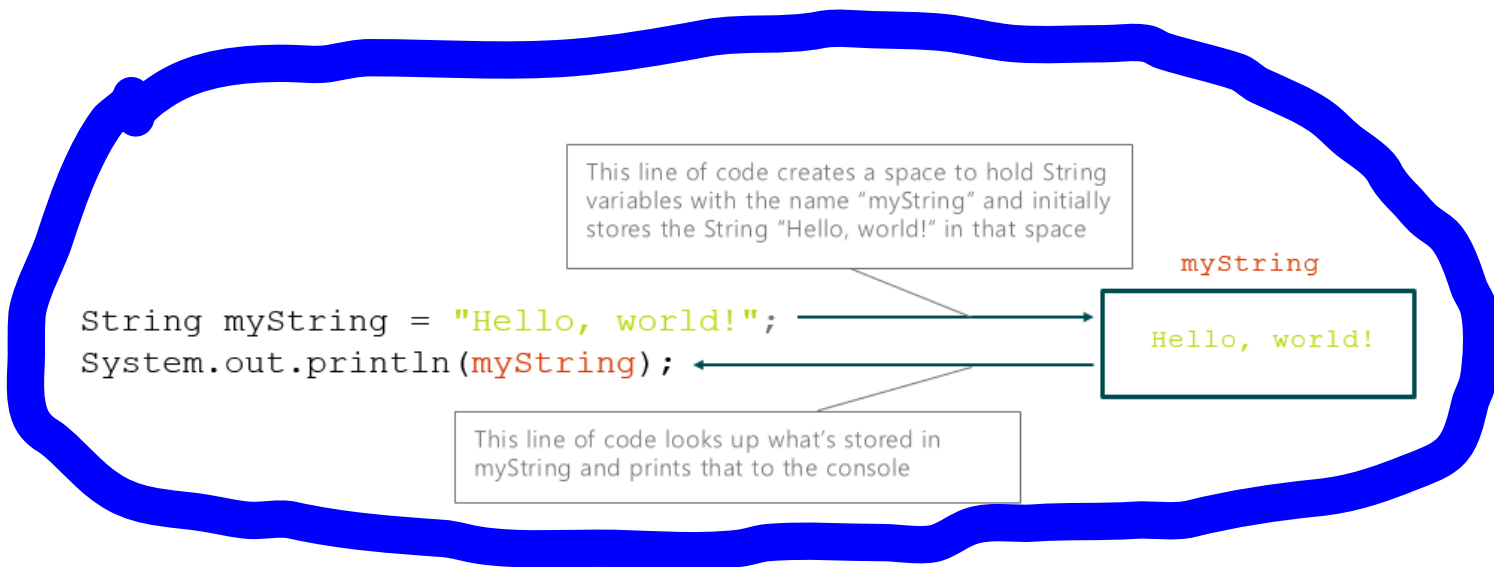
```
myString = "Hello yourself!";
```

```
System.out.println(myString);
```

Output:

Hello, world!

Hello yourself!



You can only declare a variable once. If you try to declare it a second time, Java will report it as an error. This also means you can only use the name of a variable once within a single scope.

Scope

Any variable you declare between the curly braces of a method is called a **local variable** because you can only use it locally inside the method. If you try to use it outside the method, Java will report an error, "cannot find symbol". In general, we say that a variable's **scope** is the lines of code where it can be used, the places in your code where it is "visible".

You can, however, declare two different variables that use the same name in two different methods! Consider this code:

```
public class Song {  
  
    public static void main (String[] args) {  
  
        String line = "This is the chorus";  
  
        System.out.println(line);  
  
        verse();  
  
        System.out.println(line);  
    }  
  
    public static void verse() {  
  
        String line = "This is my verse";
```

```
        System.out.println(line);  
    }  
}
```

This program has two variables named "line", but one is local to the verse method and can only be used there, and the other is local to the main method, and can only be used there. So, the program prints:

chorus

verse

chorus

Class constants

In addition to declaring local variables inside a method, you can also declare variables that are NOT part of any method, but whose scope is the entire class – these are called, not surprisingly, **class variables**. For now, we will use these only for declaring variables whose value never changes – called “**class constants**”.

To declare one of these, you need to add the terms `public static final` before the type, AND set the value as you would in an assignment statement. **It's best to place these right below the class declaration like this:**

```
public class myClass {  
  
    public static final double PI = 3.14;  
  
    public static final int MAX_SPEED = 80;  
  
    public static final int DAYS_IN_WEEK = 7;  
  
  
    public static void main(String[] args) {  
  
        ...  
    }  
}
```

Note that the Java naming convention is to use **all capital letters for class constants**, separating words by underscores to make them readable – because remember no variable names can have spaces in them!

Why declare a variable with a fixed value? Isn't the whole point of a variable is that its value can vary? Well, yes, but there are two common and useful ways to use a class constant:

1. You want all methods in the class to use exactly the same value for something; for example, use 3.14 for PI, or MAX_SPEED for 80. If you later want to change this value, you only have to change it in one place. This not only avoids having to find everywhere it's used and change it multiple times, but, more important, ensures that you don't overlook one place or accidentally change an 80 somewhere in the program that represented a maximum age instead of a speed!
2. You want to use names rather than values so that the meaning of an expression is clear; for example, using DAYS_IN_WEEK instead of 7 as in:

```
calendarRows = DAYS_IN_WEEK;
```

It is considered good style to replace repeated numbers with a class constant whenever possible to make your code more readable.

Mathematical Operations

<<Video>>

Basic Math

You can write arithmetic expressions in Java just like you do in algebra. Addition(+), subtraction (-), multiplication (*) and division(/) all work for both int and real data types. You can use parentheses as well to force an evaluation to have a higher level or precedence. The order of operations is left to right, and the precedence is also the same as in algebra:

1. Parentheses
2. Multiply and Divide
3. Add and Subtract

Many algebra teachers summarize this as PEMDAS, but in Java, it's really PMDAS because there is no exponent operator.

Dividing two integers in Java can be a little confusing: 7 / 2 gives a result of 3, not 3.5! That's because when Java divides two integers, it drops (or "truncates") the decimal part to make the result an integer too. It does not round!

Java does provide another operator for getting the remainder: %, generally called “mod”, short for modulus. For example:

`7 % 2 == 1`

`15 % 4 == 3`

`6 % 2 == 0`

Shortcuts

Java provides some extra arithmetic operators for things that programs do frequently. They don’t do anything you can’t do with the simple +, -, *, / and % operators; they’re just quicker to write, and, once you’re fluent in Java, easier to read at a glance. These operators work with both integers and doubles:

operator	example	equivalent
++	<code>x++;</code>	<code>x = x + 1;</code>
--	<code>x--;</code>	<code>x = x - 1;</code>
+=	<code>x += y;</code>	<code>x = x + y;</code>
-=	<code>x -= y;</code>	<code>x = x - y;</code>
*=	<code>x *= y;</code>	<code>x = x * y;</code>

operator	example	equivalent
/=	x /= y;	x = x / y;
%=	x %= y;	x = x % y;

Mixed int and double expressions

Whenever Java applies an operator to two int values or variables, the result is **also of type int**; when it's two doubles, the result is **always a double**. But, if you use an operator with **one int and one double**, the **result is always a double**, and it doesn't matter which comes first. This makes sense since every int can be represented as a double, but **certainly not vice versa!** So, all of these evaluate to 1.5:

`3.0 / 2 == 1.5`

`3 / 2.0 == 1.5`

`3.0 / 2.0 == 1.5`

And, all of these evaluate to 2.0 (a double), not 2 (an int):

`3.0 - 1 == 2.0`

`3 - 1.0 == 2.0`

`3.0 - 1.0 == 2.0`

This seems very simple, but when you combine it with the order of operations, it can become confusing. For example,

`7 / 2 * 2.0 == 6.0`

BUT

`2.0 * 7 / 2 == 7.0`

In the first, Java divides the two integers 7/2 and gets the integer 3, and then multiplies the integer by the double 2.0 to get a double 6.0. In the second, it multiplies the double 2.0 by 7 and gets the double 14.0, and then divides that double by the integer 2 and gets the double 7.0. The order in which you write an expression matters a lot when using a combination of integers and doubles!

Here are some examples of how the computer would evaluate some expressions that combine types, step by step:

$$\begin{array}{rcl}
 2.0 * 2.4 + 2.25 * 4.0 / 2.0 & & \\
 \swarrow \quad \searrow & & \\
 4.8 & + & 2.25 * 4.0 / 2.0 \\
 & & \swarrow \quad \searrow \\
 4.8 & + & 9.0 / 2.0 \\
 & & \swarrow \quad \searrow \\
 4.8 & + & 4.5 \\
 \swarrow \quad \searrow & & \\
 9.3 & &
 \end{array}$$

$$\begin{array}{rcl}
 7 / 3 * 1.2 + 3 / 2 & & \\
 \swarrow \quad \searrow & & \\
 2 & * & 1.2 + 3 / 2 \\
 \swarrow \quad \searrow & & \\
 2.4 & + & 3 / 2 \\
 & & \swarrow \quad \searrow \\
 2.4 & + & 1.5 \\
 \swarrow \quad \searrow & & \\
 3.9 & &
 \end{array}$$

$$\begin{array}{rcl}
 2.0 + 10 / 3 * 2.5 - 6 / 4 & & \\
 & \swarrow \quad \searrow & \\
 2.0 + & 3 & * 2.5 - 6 / 4 \\
 & \swarrow \quad \searrow & \\
 2.0 + & 7.5 & - 6 / 4 \\
 & & \swarrow \quad \searrow \\
 2.0 + & 7.5 & - 1.5 \\
 \swarrow \quad \searrow & & \\
 9.5 & - & 1.5 \\
 \swarrow \quad \searrow & & \\
 8.0 & &
 \end{array}$$

String Concatenation

<<Video>>

One of the arithmetic operators can also be used on Strings: the plus sign ("**+**"). When used with two Strings, it simply "**concatenates**" (connects) the two to make a single new String with all the characters of the first, followed by all the characters in the second. For example,

Expression	Result	Data Type
123 + 10	133	int
"123" + "10"	"12310"	String

When you concatenate a String, including an empty String, with any other data type, Java returns a String:



Expression	Value
"123" + 10	"12310"
10.6 + "123"	"10.6123"
"" + 10	"10"

Expression	Value
"123" + '1'	"1231"
"is " + true	"is true"

Note that if you are combining mathematical operations and Strings Java will evaluate the expression from left to right, converting between types as appropriate. So the following expression:



```
1 + 2 + "3" + 4 + 5
```

... results in:

```
"3345"
```

However, the order of operations applies when combining mathematical operations and concatenation, like so:

```
1 * 2 + "3" + 4 * 5 = "2320"
```

If you're concatenating two words or similar values you may need to add a String with a space character between them:

```
String firstName = "John";
```

```
String lastName = "Doe";
```

```
System.out.println(var1 + var2);
```

```
System.out.println(var1 + " " + var2);
```

... results in:

```
JohnDoe
```

```
John Doe
```

Casting

As the last two sections showed, you need to be careful when combining values of different data types. Java lets you explicitly change data from type to another or "**cast**" the data to a different type. You do this by expressly calling out what you want your data type to be after evaluation:

(resulting data type) expression;

The most common use is to cast a double into an integer for rounding the result of an integer by integer division instead of truncating it:

Expression	Value	Comment
10/4	2	a more accurate arithmetic result is 2.5, but as both numbers are ints the result is an int and the decimal is truncated
10/4.0	2.5	result is a double, so decimal is included
(int)(10/4.0)	2	takes double result of 2.5 and turns it into an int by dropping the decimal
(int)10/4.0	2.5	Without the parenthesis the 10 is cast to an int, even though it already is. The result is the int 10 / 4.0 who's result is 2.5

As the last expression shows, casting has higher precedence than any other operator (except parentheses, of course), and only cast the value immediately to its right.

You can also cast an int to a double, or accomplish the same thing by combining the types:

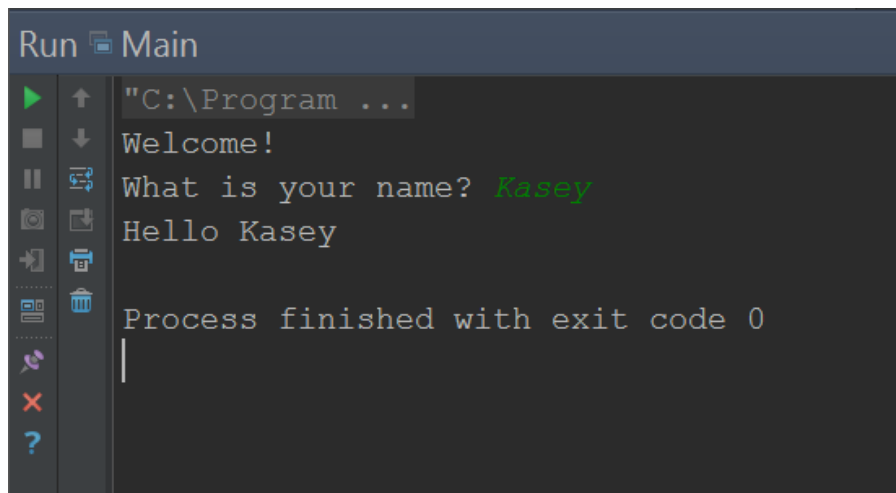
Expression	Value	Comment
(double)10	10.0	cast int to double
10*1.0	10.0	combine int with double

Interactive Programs

Scanner

<<Video>>

So far we've used `System.out.println` (and `.print`) to make your Java programs write output text to the console. Now we need to learn how to let you type on the console to provide input text to your program. Since most programs are used by someone other than the programmer who wrote it, we often refer to this input as coming from "the user". An "interactive program" is one that gets input from the user – not just the programmer! – that controls its operation.



There are lots of different ways to get input from users – keyboards, mice, touch screens, speech, etc. Java uses different classes of objects to handle this variety of different input mechanisms. The object we will use for the console keyboard is called "**Scanner**" and it is designed to handle text input. It is not built in, so we need to tell Java to look for it in the util library with this special line at the beginning of our program:

```
import java.util.*;
```

Then, typically in a method, we create our own Scanner object with a name we define, like "input" and tell it to read text from `System.in` (the equivalent of the `System.out` we use for output):

```
Scanner input = new Scanner(System.in);
```

Finally, we tell our object to get the next complete line of input from the user into a String variable:

```
String name = input.nextLine();
```


Whew! That's a lot more complicated than output with `println()`! It's also a lot more powerful, and these three lines will make much more sense once we put them all together in a program.

LearnScanner.java

```
import java.util.Scanner;

public class LearnScanner {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        System.out.println("Welcome!");

        System.out.print("What is your name?");

        String name = input.next();

        System.out.println("Hello " + name);

    }

}
```

The above code creates a new scanner and allows the user to type in any String. When Java gets to the line `input.next()` it pauses and waits for the user to type something into the console and hit "enter", then it picks back up where it left off. In this case you can type in any String without white space and it will be stored in the variable "name".

What if you want to get an int or a double instead of a String? Easy:

```
int count = input.nextInt();

double amount = input.nextDouble();
```

These allow the user to enter in numbers that you then save in the appropriate data types. Maybe you want to get some more information about your user you could read in multiple items like this:

```
String name = input.next();
```

```
int age = input.nextInt();
```

```
double weight = input.nextDouble();
```

```
System.out.println(name + " is " + age + " year old and weighs " + weight + "kg")
```

The one drawback to this simple use of a Scanner is that if you expect the user to enter an integer, and then enter something else, the program will immediately end with a cryptic error message like:

```
Exception in thread "main" java.util.InputMismatchException
```

The same problem happens with a double, although the scanner will happily accept an integer input and convert it to a double just like it does in an assignment statement. There are ways of dealing with this cleanly, but we're going to trust the user to enter the right things for now!

You can also let the user enter in as much as they like, including spaces by using the `nextLine` method. This will return a `String` of everything the user types before they hit "enter".

```
String line = input.nextLine();
```